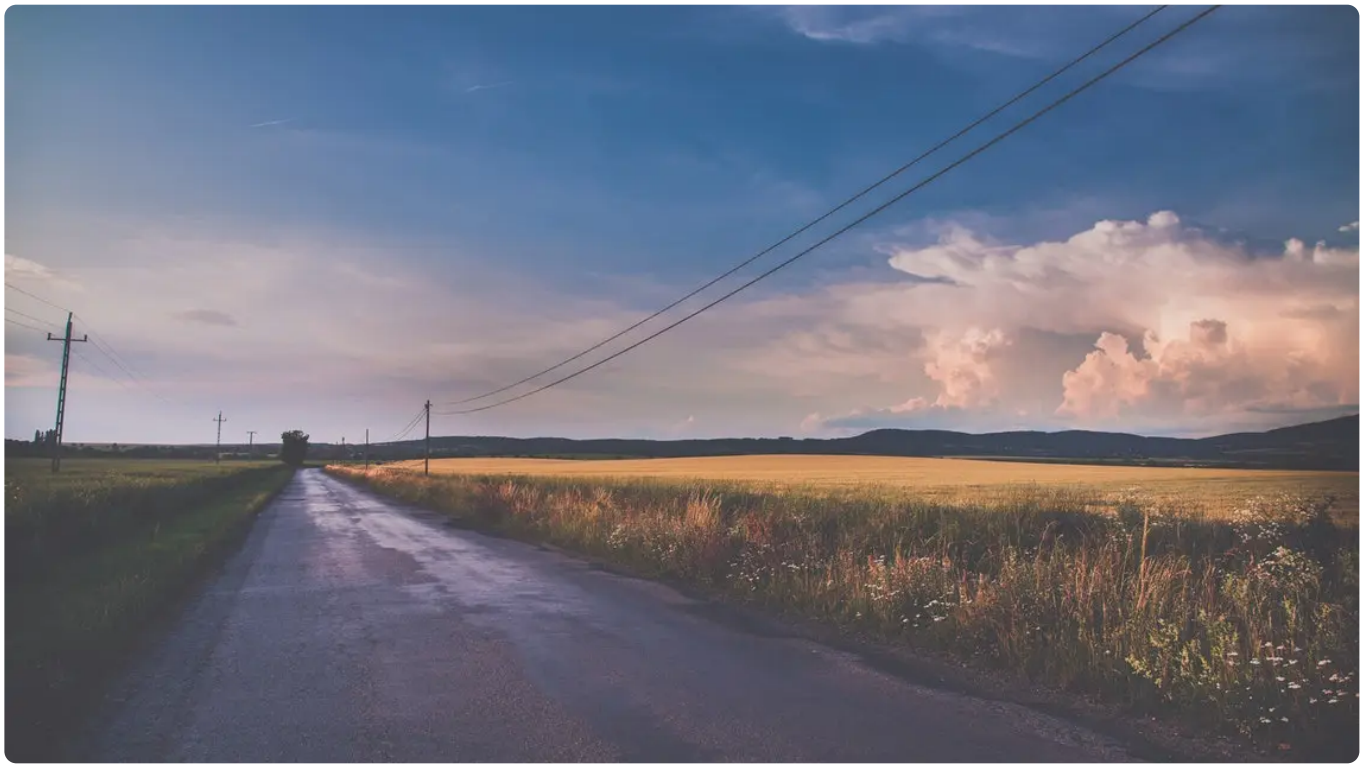
下载APP

# 20 | 控制结构：Go中的switch语句有哪些变化？

2021-11-26 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >



讲述：Tony Bai
时长 17:59 大小 16.48M

你好，我是 Tony Bai。

经过前两节课的学习，我们已经掌握了控制结构中的分支结构以及循环结构。前面我们也提到过，在计算机世界中，再复杂的算法都可以通过顺序、分支和循环这三种基本的控制结构构造出来。所以，理论上讲，我们现在已经具备了实现任何算法的能力了。

不过理论归理论，我们还是要回到现实中来，继续学习 Go 语言中的控制结构，现在我们还差一种分支控制结构没讲。除了 if 语句之外，Go 语言还提供了一种更适合多路分支执行的分支控制结构，也就是 **switch 语句**。

今天这一节课，我们就来系统学习一下 switch 语句。Go 语言中的 switch 语句继承自它的先祖 C 语言，所以我们这一讲的重点是 Go switch 语句相较于 C 语言的 switch，有哪

些重要的改进与创新。

在讲解改进与创新之前，我们先来认识一下 switch 语句。

## 认识 switch 语句

我们先通过一个例子来直观地感受一下 switch 语句的优点。在一些执行分支较多的场景下，使用 switch 分支控制语句可以让代码更简洁，可读性更好。

比如下面例子中的 readByExt 函数会根据传入的文件扩展名输出不同的日志，它使用了 if 语句进行分支控制：

复制代码

```
func readByExt(ext string) {
    if ext == "json" {
        println("read json file")
    } else if ext == "jpg" || ext == "jpeg" || ext == "png" || ext == "gif" {
        println("read image file")
    } else if ext == "txt" || ext == "md" {
        println("read text file")
    } else if ext == "yml" || ext == "yaml" {
        println("read yaml file")
    } else if ext == "ini" {
        println("read ini file")
    } else {
        println("unsupported file extension:", ext)
    }
}
```

如果用 switch 改写上述例子代码，我们可以这样来写：

复制代码

```
func readByExtBySwitch(ext string) {
    switch ext {
    case "json":
        println("read json file")
    case "jpg", "jpeg", "png", "gif":
        println("read image file")
    case "txt", "md":
        println("read text file")
    case "yml", "yaml":
        println("read yaml file")
    case "ini":
```

```
        println("read ini file")
    default:
        println("unsupported file extension:", ext)
    }
}
```

从代码呈现的角度来看，针对这个例子，使用 switch 语句的实现要比 if 语句的实现更加简洁紧凑。并且，即便你这个时候还没有系统学过 switch 语句，相信你也能大致读懂上面 readByExtBySwitch 的执行逻辑。

简单来说，readByExtBySwitch 函数就是将输入参数 ext 与每个 case 语句后面的表达式做比较，如果相等，就执行这个 case 语句后面的分支，然后函数返回。这里具体的执行逻辑，我们在后面再分析，现在你有个大概认识就好了。

接下来，我们就来进入正题，来看看 Go 语言中 switch 语句的一般形式：

复制代码

```
switch initStmt; expr {
    case expr1:
        // 执行分支1
    case expr2:
        // 执行分支2
    case expr3_1, expr3_2, expr3_3:
        // 执行分支3
    case expr4:
        // 执行分支4
    ... ...
    case exprN:
        // 执行分支N
    default:
        // 执行默认分支
}
```

我们按语句顺序来分析一下。首先看这个 switch 语句一般形式中的第一行，这一行由 switch 关键字开始，它的后面通常接着一个表达式（expr），这句中的 initStmt 是一个可选的组成部分。和 if、for 语句一样，我们可以在 initStmt 中通过短变量声明定义一些在 switch 语句中使用的临时变量。

接下来，switch 后面的大括号内是一个个代码执行分支，每个分支以 case 关键字开始，每个 case 后面是一个表达式或是一个逗号分隔的表达式列表。这里还有一个以 default 关键字开始的特殊分支，被称为**默认分支**。

最后，我们再来看 switch 语句的执行流程。其实也很简单，switch 语句会用 expr 的求值结果与各个 case 中的表达式结果进行比较，如果发现匹配的 case，也就是 case 后面的表达式，或者表达式列表中任意一个表达式的求值结果与 expr 的求值结果相同，那么就会执行该 case 对应的代码分支，分支执行后，switch 语句也就结束了。如果所有 case 表达式都无法与 expr 匹配，那么程序就会执行 default 默认分支，并且结束 switch 语句。

那么问题就来了！在有多个 case 执行分支的 switch 语句中，**Go 是按什么次序对各个 case 表达式进行求值，并且与 switch 表达式（expr）进行比较的**？

我们通过一段示例代码来回答这个问题。这是一个一般形式的 switch 语句，为了能呈现 switch 语句的执行次序，我以多个输出特定日志的函数作为 switch 表达式以及各个 case 表达式：

复制代码

```
func case1() int {
    println("eval case1 expr")
    return 1
}

func case2_1() int {
    println("eval case2_1 expr")
    return 0
}
func case2_2() int {
    println("eval case2_2 expr")
    return 2
}

func case3() int {
    println("eval case3 expr")
    return 3
}

func switchexpr() int {
    println("eval switch expr")
    return 2
}

func main() {
```

```
    switch switchexpr() {
    case case1():
        println("exec case1")
    case case2_1(), case2_2():
        println("exec case2")
    case case3():
        println("exec case3")
    default:
        println("exec default")
    }
}
```

执行一下这个示例程序，我们得到如下结果：

复制代码

```
eval switch expr
eval case1 expr
eval case2_1 expr
eval case2_2 expr
exec case2
```

从输出结果中我们看到，Go 先对 switch expr 表达式进行求值，然后再按 case 语句的出现顺序，从上到下进行逐一求值。在带有表达式列表的 case 语句中，Go 会从左到右，对列表中的表达式进行求值，比如示例中的 case2_1 函数就执行于 case2_2 函数之前。

如果 switch 表达式匹配到了某个 case 表达式，那么程序就会执行这个 case 对应的代码分支，比如示例中的“exec case2”。这个分支后面的 case 表达式将不会再得到求值机会，比如示例不会执行 case3 函数。这里要注意一点，即便后面的 case 表达式求值后也能与 switch 表达式匹配上，Go 也不会继续去对这些表达式进行求值了。

除了这一点外，你还要注意 default 分支。**无论 default 分支出现在什么位置，它都只会在所有 case 都没有匹配上的情况下才会被执行的。**

不知道你有没有发现，这里其实有一个优化小技巧，考虑到 switch 语句是按照 case 出现的先后顺序对 case 表达式进行求值的，那么如果我们将匹配成功概率高的 case 表达式排在前面，就会有助于提升 switch 语句执行效率。这点对于 case 后面是表达式列表的语句同样有效，我们可以将匹配概率最高的表达式放在表达式列表的最左侧。

到这里，我们已经了解了 switch 语句的一般形式以及执行次序。有了这个基础后，接下来我们就来看看这节课重点：Go 语言的 switch 语句和它的“先祖”C 语言中的 Switch 语句相比，都有哪些优化与创新？

## switch 语句的灵活性

为方便对比，我们先来简单了解一下 C 语言中的 switch 语句。C 语言中的 switch 语句对表达式类型有限制，每个 case 语句只可以有一个表达式。而且，除非你显式使用 break 跳出，程序默认总是执行下一个 case 语句。这些特性开发人员带来了使用上的心智负担。

相较于 C 语言中 switch 语句的“死板”，Go 的 switch 语句表现出极大的灵活性，主要表现在如下几方面：

**首先，switch 语句各表达式的求值结果可以为各种类型值，只要它的类型支持比较操作就可以了。**

C 语言中，switch 语句中使用的所有表达式的求值结果只能是 int 或枚举类型，其他类型都会被 C 编译器拒绝。

Go 语言就宽容得多了，只要类型支持比较操作，都可以作为 switch 语句中的表达式类型。比如整型、布尔类型、字符串类型、复数类型、元素类型都是可比较类型的数组类型，甚至字段类型都是可比较类型的结构体类型，也可以。下面就是一个使用自定义结构体类型作为 switch 表达式类型的例子：

复制代码

```
type person struct {
    name string
    age  int
}

func main() {
    p := person{"tom", 13}
    switch p {
    case person{"tony", 33}:
        println("match tony")
    case person{"tom", 13}:
        println("match tom")
    case person{"lucy", 23}:
        println("match lucy")
    default:
```

```
16          println("no match")
17      }
18  }
```

不过，实际开发过程中，以结构体类型为 switch 表达式类型的情况并不常见，这里举这个例子仅是为了说明 Go switch 语句对各种类型支持的广泛性。

而且，当 switch 表达式的类型为布尔类型时，如果求值结果始终为 true，那么我们甚至可以省略 switch 后面的表达式，比如下面例子：

复制代码

```
// 带有initStmt语句的switch语句
switch initStmt; {
    case bool_expr1:
    case bool_expr2:
    ... ...
}

// 没有initStmt语句的switch语句
switch {
    case bool_expr1:
    case bool_expr2:
    ... ...
}
```

不过，这里要注意，在带有 initStmt 的情况下，如果我们省略 switch 表达式，那么 initStmt 后面的分号不能省略，因为 initStmt 是一个语句。

**第二点：switch 语句支持声明临时变量。**

在前面介绍 switch 语句的一般形式中，我们看到，和 if、for 等控制结构语句一样，switch 语句的 initStmt 可用来声明只在这个 switch 隐式代码块中使用的变量，这种就近声明的变量最大程度地缩小了变量的作用域。

**第三点：case 语句支持表达式列表。**

在 C 语言中，如果要让多个 case 分支的执行相同的代码逻辑，我们只能通过下面的方式实现：

复制代码

```
void check_work_day(int a) {
    switch(a) {
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
            printf("it is a work day\n");
            break;
        case 6:
        case 7:
            printf("it is a weekend day\n");
            break;
        default:
            printf("do you live on earth?\n");
    }
}
```

在上面这段 C 代码中，case 1~case 5 匹配成功后，执行的都是 case 5 中的代码逻辑，case 6~case 7 匹配成功后，执行的都是 case 7 中的代码逻辑。

之所以可以实现这样的逻辑，是因为当 C 语言中的 switch 语句匹配到某个 case 后，如果这个 case 对应的代码逻辑中没有 break 语句，那么代码将继续执行下一个 case。比如当 a = 3 时，case 3 后面的代码为空逻辑，并且没有 break 语句，那么 C 会继续向下执行 case4、case5，直到在 case 5 中调用了 break，代码执行流才离开 switch 语句。

这样看，虽然 C 也能实现多 case 语句执行同一逻辑的功能，但在 case 分支较多的情况下，代码会显得十分冗长。

Go 语言中的处理要好得多。Go 语言中，switch 语句在 case 中支持表达式列表。我们可以用表达式列表实现与上面的示例相同的处理逻辑：

复制代码

```
func checkWorkday(a int) {
    switch a {
    case 1, 2, 3, 4, 5:
        println("it is a work day")
    case 6, 7:
        println("it is a weekend day")
    default:
        println("are you live on earth")
```

```
    }
}
```

根据前面我们讲过的 switch 语句的执行次序，理解上面这个例子应该不难。和 C 语言实现相比，使用 case 表达式列表的 Go 实现简单、清晰、易懂。

**第四点：取消了默认执行下一个 case 代码逻辑的语义。**

在前面的描述和 check_work_day 这个 C 代码示例中，你都能感受到，在 C 语言中，如果匹配到的 case 对应的代码分支中没有显式调用 break 语句，那么代码将继续执行下一个 case 的代码分支，这种“隐式语义”并不符合日常算法的常规逻辑，这也经常被诟病为 C 语言的一个缺陷。要修复这个缺陷，我们只能在每个 case 执行语句中都显式调用 break。

Go 语言中的 Swith 语句就修复了 C 语言的这个缺陷，取消了默认执行下一个 case 代码逻辑的“非常规”语义，每个 case 对应的分支代码执行完后就结束 switch 语句。

如果在少数场景下，你需要执行下一个 case 的代码逻辑，你可以显式使用 Go 提供的关键字 fallthrough 来实现，这也是 Go“显式”设计哲学的一个体现。下面就是一个使用 fallthrough 的 switch 语句的例子，我们简单来看一下：

复制代码

```
func case1() int {
    println("eval case1 expr")
    return 1
}

func case2() int {
    println("eval case2 expr")
    return 2
}

func switchexpr() int {
    println("eval switch expr")
    return 1
}

func main() {
    switch switchexpr() {
    case case1():
        println("exec case1")
        fallthrough
```

```
21      case case2():
22          println("exec case2")
23          fallthrough
24      default:
25          println("exec default")
26      }
27  }
```

执行一下这个示例程序，我们得到这样的结果：

复制代码

```
1 eval switch expr
2 eval case1 expr
3 exec case1
4 exec case2
5 exec default
```

我们看到，switch expr 的求值结果与 case1 匹配成功，Go 执行了 case1 对应的代码分支。而且，由于 case1 代码分支中显式使用了 fallthrough，执行完 case1 后，代码执行流并没有离开 switch 语句，而是继续执行下一个 case，也就是 case2 的代码分支。

这里有一个注意点，由于 fallthrough 的存在，Go 不会对 case2 的表达式做求值操作，而会直接执行 case2 对应的代码分支。而且，在这里 case2 中的代码分支也显式使用了 fallthrough，于是最后一个代码分支，也就是 default 分支对应的代码也被执行了。

另外，还有一点要注意的是，如果某个 case 语句已经是 switch 语句中的最后一个 case 了，并且它的后面也没有 default 分支了，那么这个 case 中就不能再使用 fallthrough，否则编译器就会报错。

到这里，我们看到 Go 的 switch 语句不仅修复了 C 语言 switch 的缺陷，还为 Go 开发人员提供了更大的灵活性，我们可以使用更多类型表达式作为 switch 表达式类型，也可以使用 case 表达式列表简化实现逻辑，还可以自行根据需要，确定是否使用 fallthrough 关键字继续向下执行下一个 case 的代码分支。

除了这些之外，Go 语言的 switch 语句还支持求值结果为类型信息的表达式，也就是 type switch 语句，接下来我们就详细分析一下。

## type switch

“type switch”这是一种特殊的 switch 语句用法，我们通过一个例子来看一下它具体的使用形式：

复制代码

```
1  func main() {
2      var x interface{} = 13
3      switch x.(type) {
4      case nil:
5          println("x is nil")
6      case int:
7          println("the type of x is int")
8      case string:
9          println("the type of x is string")
10     case bool:
11         println("the type of x is string")
12     default:
13         println("don't support the type")
14     }
15 }
```

我们看到，这个例子中 switch 语句的形式与前面是一致的，不同的是 switch 与 case 两个关键字后面跟着的表达式。

switch 关键字后面跟着的表达式为`x.(type)`，这种表达式形式是 switch 语句专有的，而且也只能在 switch 语句中使用。这个表达式中的 **x 必须是一个接口类型变量**，表达式的求值结果是这个接口类型变量对应的动态类型。

什么是一个接口类型的动态类型呢？我们简单解释一下。以上面的代码`var x interface{} = 13`为例，x 是一个接口类型变量，它的静态类型为`interface{}`，如果我们将整型值 13 赋值给 x，x 这个接口变量的动态类型就为 int。关于接口类型变量的动态类型，我们后面还会详细讲，这里先简单了解一下就可以了。

接着，case 关键字后面接的就不是普通意义上的表达式了，而是一个个具体的类型。这样，Go 就能使用变量 x 的动态类型与各个 case 中的类型进行匹配，之后的逻辑就都是一样的了。

现在我们运行上面示例程序，输出了 x 的动态变量类型：

复制代码

```
the type of x is int
```

不过，通过x.(type)，我们除了可以获得变量 x 的动态类型信息之外，也能获得其动态类型对应的值信息，现在我们把上面的例子改造一下：

复制代码

```
func main() {
    var x interface{} = 13
    switch v := x.(type) {
    case nil:
        println("v is nil")
    case int:
        println("the type of v is int, v =", v)
    case string:
        println("the type of v is string, v =", v)
    case bool:
        println("the type of v is bool, v =", v)
    default:
        println("don't support the type")
    }
}
```

这里我们将 switch 后面的表达式由x.(type)换成了v := x.(type)。对于后者，你千万不要认为变量 v 存储的是类型信息，其实 **v 存储的是变量 x 的动态类型对应的值信息**，这样我们在接下来的 case 执行路径中就可以使用变量 v 中的值信息了。

然后，我们运行上面示例，可以得到 v 的动态类型和值：

复制代码

```
the type of v is int, v = 13
```

另外，你可以发现，在前面的 type switch 演示示例中，我们一直使用 interface{}这种接口类型的变量，Go 中所有类型都实现了 interface{}类型，所以 case 后面可以是任意类型信息。

但如果在 switch 后面使用了某个特定的接口类型 I，那么 case 后面就只能使用实现了接口类型 I 的类型了，否则 Go 编译器会报错。你可以看看这个例子：

复制代码

```
type I interface {
    M()
}

type T struct {
}

func (T) M() {
}

func main() {
    var t T
    var i I = t
    switch i.(type) {
    case T:
        println("it is type T")
    case int:
        println("it is type int")
    case string:
        println("it is type string")
    }
}
```

在这个例子中，我们在 type switch 中使用了自定义的接口类型 I。那么，理论上所有 case 后面的类型都只能是实现了接口 I 的类型。但在这段代码中，只有类型 T 实现了接口类型 I，Go 原生类型 int 与 string 都没有实现接口 I，于是在编译上述代码时，编译器会报出如下错误信息：

复制代码

```
19:2: impossible type switch case: i (type I) cannot have dynamic type int (mi
21:2: impossible type switch case: i (type I) cannot have dynamic type string
```

好了，到这里，关于 switch 语句语法层面的知识就都学习完了。Go 对 switch 语句的优化与增强使得我们在日常使用 switch 时很少遇到坑，但这也并不意味着没有，最后我们就来看在 Go 编码过程中，我们可能遇到的一个与 switch 使用有关的问题，跳不出循环的 break。

## 跳不出循环的 break

在上一节课讲解 break 语句的时候，我们曾举了一个找出整型切片中第一个偶数的例子，当时我们是把 for 与 if 语句结合起来实现的。现在，我们把那个例子中 if 分支结构换成这节课学习的 switch 分支结构试试看。我们这里直接看改造后的代码：

复制代码

```
func main() {
    var sl = []int{5, 19, 6, 3, 8, 12}
    var firstEven int = -1

    // find first even number of the interger slice
    for i := 0; i < len(sl); i++ {
        switch sl[i] % 2 {
        case 0:
            firstEven = sl[i]
            break
        case 1:
            // do nothing
        }
    }
    println(firstEven)
}
```

我们运行一下这个修改后的程序，得到结果为 12。

奇怪，这个输出的值与我们的预期的好像不太一样。这段代码中，切片中的第一个偶数是 6，而输出的结果却成了切片的最后一个偶数 12。为什么会出现这种结果呢？

这就是 Go 中 break 语句与 switch 分支结合使用会出现一个“小坑”。和我们习惯的 C 家族语言中的 break 不同，Go 语言规范中明确规定，**不带 label 的 break 语句中断执行并跳出的，是同一函数内 break 语句所在的最内层的 for、switch 或 select**。所以，上面这个例子的 break 语句实际上只跳出了 switch 语句，并没有跳出外层的 for 循环，这也就是程序未按我们预期执行的原因。

要修正这一问题，我们可以利用上节课学到的带 label 的 break 语句试试。这里我们也直接看看改进后的代码:

复制代码

```
func main() {
```

```
    var sl = []int{5, 19, 6, 3, 8, 12}
    var firstEven int = -1

    // find first even number of the interger slice
loop:
    for i := 0; i < len(sl); i++ {
        switch sl[i] % 2 {
        case 0:
            firstEven = sl[i]
            break loop
        case 1:
            // do nothing
        }
    }
    println(firstEven) // 6
}
```

在改进后的例子中，我们定义了一个 label：loop，这个 label 附在 for 循环的外面，指代 for 循环的执行。当代码执行到“break loop”时，程序将停止 label loop 所指代的 for 循环的执行。关于带有 label 的 break 语句，你可以再回顾一下第 19 讲，这里就不多说了。

和 switch 语句一样能阻拦 break 跳出的还有一个语句，那就是 select，我们后面讲解并发程序设计的时候再来详细分析。

## 小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我们讲解了 Go 语言提供的另一种分支控制结构：switch 语句。和 if 分支语句相比，在一些执行分支较多的场景下，使用 switch 分支控制语句可以让代码更简洁、可读性更好。

Go 语言的 switch 语句继承自 C 语言，但“青出于蓝而胜于蓝”，Go 不但修正了 C 语言中 switch 语句默认执行下一个 case 的“坑”，还对 switch 语句进行了改进与创新，包括支持更多类型、支持表达式列表等，让 switch 的表达力得到进一步提升。

除了使用常规表达式作为 switch 表达式和 case 表达式之外，Go switch 语句又创新性地支持 type switch，也就是用类型信息作为分支条件判断的操作数。在 Go 中，这种使用方

式也是 switch 所独有的。这里，我们要注意的是只有接口类型变量才能使用 type switch，并且所有 case 语句中的类型必须实现 switch 关键字后面变量的接口类型。

最后还需要你记住的是 switch 会阻拦 break 语句跳出 for 循环，就像我们这节课最后那个例子中那样，对于初学者来说，这是一个很容易掉下去的坑，你一定不要走弯路。

## 思考题

为了验证在多分支下基于 switch 语句实现的分支控制更为简洁，你可以尝试将这节课中的那些稍复杂一点的例子，改写为基于 if 条件分支的实现，然后再对比一下两种实现的复杂性，直观体会一下 switch 语句的优点。

欢迎你把这节课分享给更多对 Go 语言中的 switch 语句感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 8 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 控制结构：Go的for循环，仅此一种

下一篇 21 | 函数：请叫我“一等公民”

更多学习推荐

## 精选留言 (6)

写留言

**aoe**
2021-11-26

今天《极客时间》两个专栏同时更新，主题都是 switch

- 《深入剖析 Java 新特性》06 | Switch表达式：怎么简化多情景操作？
- 《Tony Bai · Go 语言第一课》20 | 控制结构：Go中的switch语句有哪些变化？

...

展开

4

**lesserror**
2021-11-26

Tony Bai 老师的文章讲解的非常细致，鼓掌。

想问一下老师，文中的内容基本都能理解，但是过一段时间就遗忘比较多了，尤其是后面的内容涉及到前面的知识时。希望老师在这门课中搞个小型的实战项目，能把前面的知识串在一起就好了。...

展开

作者回复: 后续会有实战项目，尽量串联吧。

共 2 条评论 >

1

**进化菌**

2021-11-26

所以，switch 不需要 break 是出于大多数情况 switch 只需要走一条分支的缘故吗？

1

**qinsi**

2021-12-04

type switch里的

switch v := x.(type) {

看上去像是一个初始化语句，但其实是一个type guard，所以后面没有分号。如果有初始...

展开

**Bynow**

2021-11-26

内容查漏补缺，支持老师

展开

**罗杰**

2021-11-26

讲的非常详细，值得好好学习

展开